Handystats Documentation

Release 1.5.0

Danil Osherov

July 07, 2015

Contents

Handystats is C++ library for collecting **user-defined in-process runtime statistics**.

Handystats allows users to monitor their **multithreaded applications** in a **production environment** with **low overhead**.

> **Caution:** Handystats library is in **"beta" phase**. Everything may be changed!

Contents:

CHAPTER 1

About

1.1 What Is Handystats?

Handystats is C++ library for collecting **user-defined in-process runtime statistics** with **low overhead** that allows users to monitor their applications in a production environment.

What Does It Mean, Exactly?

- By **statistics** we mean a collection of metrics (counters, timers, gauges, etc.) and its cumulative data such as min, max, sum, count, etc.
- By **runtime** we mean that we're aimed to collect and provide statistics about program's execution in "near real-time".
- By **in-process** we mean that statistics' collection and aggregation are performed within the program's process (but still in separate thread).
- By **user-defined** we mean that you must instrument your own code with metrics that should be collected.
- By **low overhead** we mean that library's influence on application's performance must be minimal.

1.2 What Problem We Are Aimed To Solve?

Our main goal is to provide you with an ability to "*look inside*" your own program at runtime via collected statistics (which then can be displayed as numbers, graphics, etc.) and thus better understand what is going on.

Also we can provide your program with gathered statistics that can be used to adjust its own behaviour at runtime.

1.3 Motivation

Imaging how simple life would be if all programs were black boxes with number of inputs and outputs that just work. But it's almost never true despite any number of testing stages that this black box had passed. Thus administrator and/or programmer should have to know what and why is happening inside that black box.

Logs could help with that, but logs are *raw data* which means someone should analyze them to produce meaningful information. And we're not mentioning their disk space consumption.

At this point runtime statistics collected and accumulated *inside* the program looks as more suitable solution. Programmer could implement counters, timers and other metrics by hand, but it's a fair amount of work considering such requirements as low overhead and support for multi-threaded program. And this should definitely not be implemented again every time one starts new project.

1.4 Inspiring Example

Consider a program that processes input events and sends processing result further:

```
void process(event_type event) {
    result = internal_process(event);
    send_process_result(result);
}
```

Here is the black box! We know neither input events count and rate nor processing and sending timings. With slightly modifications we can tell handystats library to collect such metrics:

```
void process(event_type event) {
    HANDY_COUNTER_INCREMENT("events count", 1);

    HANDY_TIMER_START("internal process time");
    result = internal_process(event);
    HANDY_TIMER_STOP("internal process time");

    HANDY_TIMER_START("send result time");
    send_process_result(result);
    HANDY_TIMER_STOP("send result time");
}
```

Here we ask handystats library to increment counter with name "`events count`" by 1 on each new event. Measurement of `internal_process` and `send_process_result` running times is performed by starting and stopping coresponding timers.

Now handystats library will collect statistics about these metrics. Then we can request collected statistics in, for example, JSON format:

```
{
    "events count": {
        "type": "counter",
        "timestamp": ...,
        "value": 10000,
        "values": {
            "min": 0,
            "max": 10000,
            "mean": 5000,
            ...
        },
        "incr-deltas": {
            ...
        },
        "decr-deltas": {
            ...
        },
        ...
    },
    "internal process time": {
        "type": "timer",
        "timestamp": ...,
        "value": 300,
        "values": {
            "min": 230,
            "max": 560,
            "mean": 320.56,
            ...
```

```
        },
        ...
    },
    "send result time": {
        ...
    },
    ...
}
```

Here “events count” counter entry contains information about:

- its type
- last timestamp
- last value
- cumulative statistics about all counter values, increasing deltas, decreasing deltas
- ...

And “internal process time” as well as “send result time” timers entries contain information about:

- its type
- last timestamp
- last value
- cumulative statistics about all timer values
- ...

CHAPTER 2

Architecture

Handystats library consists of the following components:

- metrics
- measuring points
- event message queue
- processing core
- metrics and JSON dumps

Here is an example of relation between user's application and handystats library:

2.1 Metrics

Metrics is the base of handystats library statistics.

You're able to use metrics objects locally in your application, but our main and preferable way of handling metrics is via measuring points.

See Metrics documentation for more details.

2.2 Measuring Points

Measuring points are the way to handle metrics resided inside handystats library core by passing event messages to them.

Examples of measuring points are:

```
HANDY_TIMER_START("timer");

HANDY_COUNTER_INCREMENT("counter", 1);
```

See Measuring Points documentation for more details.

2.3 Event Message Queue

Events describe actions that should be taken on metrics stored inside handystats library core. Such events are passed to the handystats library core by measuring points, which form certain events and push them into event message queue.

Event message queue is one-way communication channel between user's application and handystats library core. User's application with use of measuring points publishes event messages to the queue and handystats library core processes them.

Since there's single message queue it's prone to become system's bottleneck especially in multithreaded environment. To solve this problem we've designed this part of handystats library *lock-free*.

2.4 Processing Core

Handystats library core is the place where *hidden from user* work is done. Here separate processing thread receives event messages from the event message queue and appropriately updates internal metrics.

At a time it updates *metics* and *json dumps* of internal metrics state which can be accessed immediately.

2.5 Metrics And JSON Dumps

Metrics and **JSON dumps** are representation of internal metrics snap (which we call *dump*) in different formats.

Metrics dump is dump in object format (`std::map`) which can be easily used by user's application in runtime.

JSON dump is dump in JSON text representation which can be printed or sended further.

CHAPTER 3

Incremental Statistics

Incremental statistics is the base of handystats metrics' data aggregation mechanism. Thus understanding this type of statistics, its pros and cons is crucial.

3.1 Definition

Incremental statistics are such statistics over dataset of pairs `(value, timestamp)` that can be maintained and updated on each new data with constant time and space complexity.

Examples of incremental statistics are:

- min
- max
- sum
- moving average

One of the main properties of incremental statistics is that no raw input data have to be stored. This could be disadvantage if you would like to retrieve some special statistics from input data that are not presented in handystats library. But this approach allows us to keep memory consumption low.

3.2 Interval Statistics

Some simple incremental statistics such as *min*, *max*, *sum*, *count* and *mean* act on complete dataset. While knowing total count or sum is important it could become less meaningful in long-running programs where statistics on *recent data* (e.g., last N values or last second) is needed. Examples of such statistics could be *interval count*, *sum* and *mean*.

Implementation of precise interval statistics is based on maintenance of recent data window which could lead to memory consumption problems. At this point approximation of interval statistics is the solution. But there's no such thing as a free lunch and with approximation of interval statistics we lose precision.

We base our interval statistics implementation on modification of exponential smoothing technique.

3.3 Exponential Moving Average

Exponential moving average is an example of both exponential smoothing and interval statistics.

The basic formula of exponential moving average is:

$$EMA_t = \alpha \cdot x_t + (1 - \alpha) \cdot EMA_{t-1}$$

Here

- EMA_t is exponential moving average at time t
- x_{t-1} is datum at time $t - 1$
- α is smoothing factor, $0 < \alpha < 1$

The only configurable parameter in exponential moving average is smoothing factor α. Higher α values will discount older data faster.

If you are interested in just about N last data values the best choices for α would be $\frac{2}{N+1}$ and $\frac{1}{N}$. With this perception exponential moving average can be roughly perceived as an interval statistics for the last N data values.

3.4 Exponential Smoothing Technique For Time Intervals

Enhancement of exponential smoothing could be made by adjusting smoothing factor in runtime. Also to extend exponential smoothing to time intervals smoothing factor should be based on data timestamps and elapsed time between datum points.

For example, **interval sum** (for last time interval T) can be maintained in the following general form (corner cases are not considered):

$$S_k = x_k + (1 - E) \cdot S_{k-1}$$

Here

- S_k is *approximately* sum of data values over last time interval T that ends on x_k timestamp t_k
- x_k is datum with timestamp t_k
- E is the ratio of *elapsed* time between t_{k-1} and t_k timestamps to time interval T

The *"physical" interpretation* of interval sum formula is the following.

Let S_{k-1} be interval sum of x data values over last time interval T that ends on t_{k-1} timestamp. Also let sum of considered x values be *evenly distributed* over time interval T.

When new datum value x_k comes at timestamp t_k time interval window is shifted by $\Delta T_k = t_k - t_{k-1}$. And only $\frac{T - \Delta T_k}{T}$ part of previous time interval window overlaps with the new one.

Thus as sum of values is evenly distributed over previous time interval only $\frac{T - \Delta T_k}{T}$ part of previous interval sum should preserved. New datum value x_k contributes to the new interval sum with no modification as evenly distributed over last ΔT_k time interval.

Still this approach is approximation of precise interval statistics and might behave poorly on irregular and sparse data series, but in general its behaviour is acceptable.

Interval count statistics is easily obtained from interval sum by replacing x values with constant 1.

And **interval mean** is just the ratio of interval sum to interval count.

3.5 List of Handystats' Incremental Statistics

Here is the full list of incremental statistics that are supported by handystats library:

- min
- max
- sum
- count
- min
- moving average (exponential)
- interval count
- interval sum
- interval mean

3.6 Incremental Statistics Implementation

We base our incremental statistics implementation on Boost.Accumulators with additional moving average and interval statistics.

CHAPTER 4

Metrics

We see **metrics** and **statistics** on different levels of abstraction. While *statistics* (in particular, *Incremental Statistics*) characterize time-stamped data series whitout any relations with another series, *metrics* are data sources for statistics and can aggragate related statistics.

Mertics are an interface to represent internal processes and inherent data in user's application. And metrics' collected statistics provide user an opportunity to understand application's state.

Examples of metrics presented in handystats library are:

- counters
- timers
- gauges

4.1 Counters

Counter is just an integer that starts with initial value (often it's zero) and can be incremented and decremented by delta values.

Counters depsite and due to its simplicity are one of the most widely used type of metrics.

For example, it can represent number of active users, size of internal queue in user's application and a lot more other data with the only restriction that data change by deltas. If you want to track data that change by its value see *Gauges*.

What Statistics Counter Can Provide?

It's obvious that *last value* of counter is pretty much meaningless and gives your no information about counter's previous states and its behaviour.

Therefore, in addition to **last value** and **last timestamp** of counter we provide the following statistics:

- **values**

  This statistics collect values through which counter passes in between its changes by deltas.

  For example, if counter starts with 0 as initial value and then changes by the following deltas: `1, 2, -3, 5, 10`.

  Then values through which the counter has passed will be: `0, 1, 3, 0, 5, 15`.

- **increasing deltas**

  This statistics collect only positive deltas by which counter changeses.

  As in above example, positive deltas will be: `1, 2, 5, 10`.

- **decreasing deltas**

  This statistics collect absolute values of only negative deltas by which counter changeses.

  As in above example, positive deltas will be: `3`.

- **deltas**

  This statistics collect all deltas by which counter changeses.

  As in above example, deltas will be: `1, 2, -3, 5, 10`.

Aggregation of all of the above data (values, increasing deltas, decreasing deltas, deltas) are performed by incremental statistcs. See *Incremental Statistics* for more details.

4.2 Timers

Timer allows you to measure time spans durations and collects statistics about these duration values.

Time spans measured by particular timer can *overlap* with each other. For example, it can be time spans measured from different threads or time spans between events' entrance and its processing completion.

See chrono for more details on how we measure time spans and why it's *highly accurate*.

Timer's Instances

To support overlapping time spans by single timer we introduce the term of **timer's instance**.

Timer's instance marked with *unique ID within single timer* defines currently active time span. When this active time span ends timer's instance measures total time span's duration and updates collected timer statistics. After that corresponding timer's instance is removed from the timer and its unique ID becomes free for subsequent use.

Timer's Timeout

Sometimes user might forget to send final event for particular timer's instance or measured event took too long time for completion.

Anyway, we can't allow timer's instance to live forever.

Timer's instance is allowed to live with no events for this particular instance up to **idle timeout**. If no events have been received by timer's instance during this time interval, timer's instance is considered dead and is removed from the timer with no impact on collected statistics.

You can configure this option, see *Configuration* for more details.

But if you want to extend lifetime of particular timer's instance you should send **heartbeat events** during measured time interval to tell handystats library core that timer's instance is alive.

What Statistics Timer Can Provide?

As with counter's statistics only **last time span's duration** and **last timestamp** are not enough.

Thus handystats library provide statistics about all **values** collected by timer – durations of all correctly finished timer's instances.

Aggregation of collected values are performed by incremental statistics. See *Incremental Statistics* for more details.

4.3 Gauges

Gauge is the simplest type of statistics. It collects only *values* that you pass to the metric.

For example, you might want to measure size of some structure that updates without your influence. At this point you can request size of the structure from time to time and pass requested values to the gauge metric.

Underlying data aggregation is exactly incremental statistics. See *Incremental Statistics* for more details.

CHAPTER 5

Configuration

Handystats library may suit your needs with its default behaviour, but you might want to tune its configuration for your specific case.

You can pass your configuration options **before** starting handystats library core (`HANDY_INIT()` call) with the following methods:

- `HANDY_CONFIGURATION_FILE` – accepts file name with configuration data in JSON format.

  Example:

```
HANDY_CONFIGURATION_FILE("handystats.cfg");
```

  and `handystats.cfg` file's content:

```
{
    "handystats": {
        "timer": {
            "idle-timeout": 100
        },
        "message-queue": {
            "sleep-on-empty": [1, 10, 100, 1000]
        }
    }
}
```

- `HANDY_CONFIGURATION_JSON` – accepts string with configuration data.

  Example:

```
HANDY_CONFIGURATION_JSON("\
        {\
            "handystats": {\
                "incremental-statistics": {\
                    "moving-interval": 1000\
                },\
                "timer": {\
                    "idle-timeout": 1500\
                },\
                "json-dump": {\
                    "interval": 1000\
                }\
            }\
        }\
    ");
```

- `HANDY_CONFIGURATION_JSON` – accepts `rapidjson::Value` object with configuration data.

In all of these methods accepted configuration data must be in JSON format.

Here is an example of such JSON with all acceptable configuration options that should be considered as skeleton for your own configuration:

```
{
    "handystats": {
        "incremental-statistics": {
            "moving-average-alpha": <double value>,
            "moving-interval": <value in msec>
        },
        "timer": {
            "idle-timeout": <value in msec>
        },
        "json-dump": {
            "interval": <value in msec>
        },
        "metrics-dump": {
            "interval": <value in msec>
        },
        "message-queue": {
            "sleep-on-empty": [<first sleep interval in usec>, <second sleep interval in usec>, ...]
        }
    }
}
```

5.1 Incremental Statistics Configuration

Following options should be specified within `"incremental-statistics"` handystats' configuration JSON entry. As an example:

```
{
    "handystats": {
        "incremental-statistics": {
            "moving-average-alpha": 0.25,
            "moving-interval": 1500
        }
    }
}
```

Read *Incremental Statistics* documentation for the backgroud of the following options.

moving-average-alpha Indirectly specifies data window length for moving average statistics.

If you want *moving average* statistic to handle approximately last N values recommended choices would be $\frac{1}{N}$ and $\frac{2}{N+1}$.

Default: 0.125

moving-interval Specifies moving time widow length in *milliseconds* over which interval count, sum and mean statistics are calculated.

Default: 1000

5.2 Timer Metric Configuration

Following options should be specified within `"timer"` handystats' configuration JSON entry. As an example:

```
{
    "handystats": {
        "timer": {
            "idle-timeout": 5000
        }
    }
}
```

Read timer-metric documentation for the backgroud of the following options.

idle-timeout Specifies time interval in *milliseconds* for which timer's instance is considered to be alive.

If no events for timer's instance have been recieved during this time interval timer's instance will be removed with no impact on collected statistics.

Default: 10000

5.3 JSON Dump Configuration

Following options should be specified within `"json-dump"` handystats' configuration JSON entry. As an example:

```
{
    "handystats": {
        "json-dump": {
            "interval": 1000
        }
    }
}
```

Read json-dump documentation for the backgroud of the following options.

interval Specifies time interval in *milliseconds* for generating JSON dump of all collected statistics.

Zero value disables JSON dump generation.

Default: 500

5.4 Metrics Dump Configuration

Following options should be specified within `"metrics-dump"` handystats' configuration JSON entry. As an example:

```
{
    "handystats": {
        "metrics-dump": {
            "interval": 1000
        }
    }
}
```

Read metrics-dump documentation for the backgroud of the following options.

interval Specifies time interval in *milliseconds* for generating metrics dump of all collected statistics.

Zero value disables metrics dump generation.

Default: 500

5.5 Message Queue Configuration

Following options should be specified within "message-queue" handystats' configuration JSON entry. As an example:

```
{
    "handystats": {
        "message-queue": {
            "sleep-on-empty": [1, 2, 4, 8, 16]
        }
    }
}
```

Read message-queue documentation for the backgroud of the following options.

sleep-on-empty Specifies sequence of time interval in *microseconds* for which handystats core's processing thread will sleep if no event messages are passed to the handystats core.

Default: [1, 5, 10, 50, 100, 500, 1000, 5000, 10000]

CHAPTER 6

Time Measurement

Fast and **accurate** elapsed time measurement is an essential part of any performance monitoring system, espesially system that is aimed to run in a production environment.

Unfortunately, there's no the one true solution for time measurement problem. Various types of operation system and processor architecture provide different methods with its pluses and minuses.

6.1 Clock Sources

Time measurement is based on an ability to get current time stamp in arbitrary unit (for example, millisecons or the number of cycles).

On POSIX systems three the most widely used sources for such "time stamps" are Time Stamp Counter, High Precision Event Timer and ACPI Power Management Timer.

6.1.1 Time Stamp Counter

One of the most fast and accurate approach to measure elapsed time is by using Time Stamp Counter.

The Time Stamp Counter (TSC) is a special 64-bit register which counts the number of processor's cycles since reset. The TSC provides the highest-resolution timing information available for that processor.

But despite its high accurancy and low overhead the TSC is hard to use now:

> The time stamp counter has, until recently, been an excellent high-resolution, low-overhead way of getting CPU timing information. With the advent of multi-core/hyper-threaded CPUs, systems with multiple CPUs, and hibernating operating systems, the TSC cannot be relied on to provide accurate results — unless great care is taken to correct the possible flaws: rate of tick and whether all cores (processors) have identical values in their time-keeping registers. There is no promise that the timestamp counters of multiple CPUs on a single motherboard will be synchronized. In such cases, programmers can only get reliable results by locking their code to a single CPU. Even then, the CPU speed may change due to power-saving measures taken by the OS or BIOS, or the system may be hibernated and later resumed (resetting the time stamp counter). In those latter cases, to stay relevant, the counter must be recalibrated periodically (according to the time resolution the application requires).

Contemporary Intel and AMD processors provide a serializing instruction `RDTSCP` to read the TSC and an indetifier indicating on which CPU the TSC was read unlike the regular instruction `RDTSC` which just reads the TSC. CPU's identifier is needed becase time stamp counters from different CPUs are not guaranteed to be synchronized.

Also an enhanced versions of TSC (`constant_tsc` and more general `invariant_tsc` and `nonstop_tsc`) are provided by modern Intel processors that run at the processor's maximum rate regardless of the actual CPU running rate.

Thus, reliance on the TSC reduces portability, as other processors may not have a similar feature and its properties. Furthermore, to convert the number of cycles to more convinient units such as millisecons TSC's rate has to be determinied accurately.

Read the following documents for more details on the Time Stamp Counter:

- TSC Synchronization Across Cores
- Game Timing and Multicore Processors
- Timestamping from the Red Hat MRG's documentation
- Wikipedia's page

6.1.2 High Precision Event Timer

HPET is a hardware timer implemented as a 64-bit up-counter counting at a frequency of at least 10 MHz.

To access the HPET's time stamp one can read it from special memory locations.

Comparing to the TSC, reading from which is, basically, reading a register from the processor, reading from the HPET clock is significantly slower for measuring time for high rate events. Thus, the Time Stamp Counter is preferred over the HPET as a clock source.

For more information about HPET see the following documents:

- Timestamping from the Red Hat MRG's documentation
- Wikipedia's page

6.1.3 ACPI Power Management Timer

APCI PM Timer is the slowest timer among specified above and is used in case of the absence of the TSC and the HPET.

See the following documents for more details:

- Timestamping from the Red Hat MRG's documentation

6.2 POSIX Clocks

POSIX clocks is a standard for implementing and representing time sources.

Nevertheless, POSIX doesn't require any particular underlying hardware clock source for implementing clocks. Thus, POSIX clocks can be seen as an abstraction over hardware clock sources that defines types of clock with its properties.

Linux supports the following types of clock that are of interest to us:

- **CLOCK_REALTIME**

  The system-wide real time (wall time) clock. This clock measures the amount of time that has elapsed since 00:00:00 January 1, 1970 Greenwich Mean Time (GMT). It can be modified by an user with the right privileges. Thus, the clock can jump forwards and backwards as the system time-of-day clock is changed, including by NTP.

- **CLOCK_MONOTONIC**

  Clock that represents monotonic time since some unspecified starting point, such as system boot. This clock is not settable by any process and is not affected by discontinous jumps in the system time, but is affected by the incremental adjustments performed by `adjtime()` and NTP.

- **CLOCK_MONOTONIC_RAW** (since Linux 2.6.32, Linux-specific)

  Similar to CLOCK_MONOTONIC, but provides access to a raw hardware-based time that is not subject to the incremental adjustments performed by `adjtime()` or NTP.

POSIX defines CLOCK_REALTIME and CLOCK_MONOTONIC types of clock, but it requires only CLOCK_REALTIME. Therefore, while Linux provides all of these clocks, on other systems analogous to CLOCK_MONOTONIC or CLOCK_MONOTONIC_RAW clocks should be used or, in case of absence of those, CLOCK_REALTIME.

Considering issues and complexity of using hardware clock sources specified above as well as for code portability using POSIX clocks is preferred for most systems.

POSIX provides the `clock_gettime()` interface for obtaining the time of a specific clock. More useful is that the function allows for nanosecond precision returning result in `timespec` structure.

For more information on POSIX clocks see the following:

- Timestamping from the Red Hat MRG's documentation
- *Linux System Programming* by Robert Love
- clock_gettime() man page

6.3 Time Intervals And Timestamps

Possible jumps of CLOCK_REALTIME make it unreliable for measuring time intervals, thus, one or the other clocks should be used instead, if available. The same way, the time from CLOCK_MONOTONIC and analogous cannot be mapped to the current real-world date and time.

This shows two conceptual different time-based problems that should be solved by performance monitoring systems:

- **measuring time intervals**

  For solving this problem any high-precision monotonic clock could be used, even hardware clock source. The only restriction is an ability to accurately convert time interval's duration from clock's units to more convinient units, such as nanoseconds.

- **timestamping events**

  This problem raises only when user wants to work with event timestamps. Inside the system any suitable clock can be used, but for the "*outside world*" timestamp should be in consensual form, such as Unix time.

6.4 Clock Concept

To hide implementation details and underlying clock type from users and other parts of the handystats library **Clock concept** similar to C++ Clock concept has been introduced:

- Type names:
  - `time_point` – represents time point from clock's epoch
  - `duration` – represents duration between time points

- Member functions:
  - `static time_point now() noexcept` – returns current time point
- Non-member functions:
  - `std::chrono::system_clock::time_point to_system_time(const clock::time_point&)` – converts `clock::time_point` to system-wide clock's time represented by `std::chrono::system_clock::time_point`

    This function should perform conversion from `clock::time_point` which may represent hardware clock source value, such as TSC, and have no connection to system-wide clock. Still, as the handystats library uses clock concept's `time_point` for both measuring time intervals and timestamping events conversion is needed for users to work with events timestamps at runtime.
  - `duration_cast` from and to `clock::duration` – performs conversion between `clock::duration` and `std::chrono::duration`

    This function should perform conversion between `clock::duration` and `std::chrono::duration`. The former may represent intervals in terms of hardware clock's ticks and have no explicit connection to convenient time units, while the latter represents time intervals in terms of convenient time units, such as `std::chrono::milliseconds`.

Note that `time_point` and `duration` types can have no connection to `std::chrono`, but this types try to follow corresponding `std::chrono` public interfaces.

The handystats library provides the following `typedef` that represents library-wide clock:

```
namespace handystats { namespace chrono {

typedef <clock concept implementation> clock;

}}
```

6.5 Implementation Details

The handystats library implements library-wide clock using the Time Stamp Counter register as the most precise and fast hardware clock source. To read the value from the TSC `RDTSCP` serializing instruction is used.

Considering specified above caveats on using the TSC we're aimed on processor architectures and operation systems that support **constant TSC** and **RDTSCP** serializing instruction.

6.5.1 The Time Stamp Counter Rate

For measuring time intervals calibration between the number of cycles and time units, specifically nanoseconds, should be performed. Thus, **the TSC's rate** should be determined.

The TSC's rate is determined by multiple interval measurements by TSC and CLOCK_MONOTONIC simultaneously and choosing median frequency between measurements. To find corresponding pair of TSC and CLOCK_MONOTONIC values at start and end of interval measurement CLOCK_MONOTONIC time retrieval is surrounded by RDTSCP calls. And pair of CLOCK_MONOTONIC time and average of TSC values is formed only if the difference between TSC values is acceptable. Otherwise, determination of corresponding pair of TSC and CLOCK_MONOTONIC values is repeated.

The TSC's rate determination should be performed at startup. To this purpose we mark the TSC initializing function with `__attribute__((constructor))` to be invoked at load-time.

Note, that `__attribute__((constructor))` is GCC-specific semantics, thereby we limit the set of supported compilers to GCC and Clang. See requirements for more details.

6.5.2 Cycles Count To System Time Conversion

To convert the number of cycles that counts from some unspecified point in time to absolute system time correlation between system time (represented by C++11's `std::chrono::system_clock` or POSIX's `CLOCK_REALTIME`) and the number of cycles should be known.

At this point there should be a "*tied*"pair of timestamp in terms of the of cycles and timestamp of system clock that correspond to the same point in time. Moreover, considering possible system time updates and adjustments the tied pair of internal and system time should be updated periodically.

Such conversion is performed by `handystats::chrono::to_system_time` function described above. Considering our focus on multithreaded applications there's no limit on the number of concurrent calls to the function, thus updates of the tied pair of internal and system time and calls to the `handystats::chrono::to_system_time` should be **thread-safe**.

To ease implementation of the update of the tied pair, let's do some math:

- let R be the rate of the TSC,
- let T_{tsc}, T_{sys} be the tied pair of internal T_{tsc} and system T_{sys} time,
- let t_{tsc} be current internal time and we want to find current system t_{sys}.

The following formula is the solution:

$$t_{sys} = T_{sys} + \frac{t_{tsc} - T_{tsc}}{R}$$

This formula can be transformed to

$$t_{sys} = \frac{t_{tsc}}{R} + (T_{sys} - \frac{T_{tsc}}{R})$$

The last term in brackets is an **offset** that fully replaces the tied pair. Thus, the only we need to update is single value instead of a pair. And such update of the offset can be performed in a **lock-free** manner.